8249-EN-01
DTIC

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188



| 1. AGENCY USE ONLY (Leave blank) | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | 7 July 1998 | 3rd interim report |

**4. TITLE AND SUBTITLE**
Conditional Estimation of Vector Patterns in Remote Sensing and GIS

**5. FUNDING NUMBERS**
N68171 97 C 9027

**6. AUTHOR(S)**
Dr.J.M.F.Masuch

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
CCSOM / PSCW / University of Amsterdam
Sarphatistraat 143
1018 GD AMSTERDAM NL

**8. PERFORMING ORGANIZATION REPORT NUMBER**
BAA III 98-1

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**
DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

**12b. DISTRIBUTION CODE**

**13. ABSTRACT (Maximum 200 words)**

Oriented patterns, such as those produced by vector propagation, raster pattern accretion, or deformation of geometric features, are common in image processing and GIS. Our approach to classifying such patterns is to decompose them into two parts: the flow field, describing the direction of anisotropy; and the residual pattern obtained by describing the image in a coordinate system built from the flow field. We develop a method for the local estimation of anisotropy and a method for combining the estimates to construct a flow coordinate system.

**14. SUBJECT TERMS**
Remote Sensing, Statistics, Artificial Intelligence, Neural Networks

**15. NUMBER OF PAGES**
29

**16. PRICE CODE**

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | | |

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

# Conditional Estimation of Vector Patterns in Remote Sensing and GIS

3nd Interim Report

Principal investigator: Dr. J.M.F. Masuch

*1 Juli 1998*

United States Army
European Research Office of the U.S. Army

USARDSG-UK, Edison House
223 Old Marylebone Road
London, NW1 5TH
England

**Contract Number:** N68171-97 C 9027

Approved for Public Release; distribution unlimited

CCSOM/Applied Logic Laboratory
PSCW - Universiteit van Amsterdam
Sarphatistraat 143
1012 GC AMSTERDAM
The Netherlands
tel: #.31.20.525 28 52
fax: #.31.20.525 28 00
e-mail: ccsoff@ccsom.uva.nl

R&D 8249-EN-01
Broad Area Announcement Proposal
submitted to the
Remote Sensing / GIS Center USACRREL
72 Lyme Road
Hanover, New Hampshire 03755 USA

1. Title

Conditional Estimation of Vector Patterns in Remote Sensing and GIS

2. Abstract

Oriented patterns, such as those produced by vector propagation, raster pattern accretion, or deformation of geometric features, are common in image processing and GIS. Our approach to classifying such patterns is to decompose them into two parts: the flow field, describing the direction of anisotropy; and the residual pattern obtained by describing the image in a coordinate system built from the flow field. We develop a method for the local estimation of anisotropy and a method for combining the estimates to construct a flow coordinate system.

3. Topical Research

A central focus in recent computational vision has been the decomposition of an intensity image into intrinsic images representing such properties as depth, reflectance, and illuminance. These intrinsic properties are believed to be more meaningful than image intensity because they describe basic independent constituents of the image formation process. Thus, for example, in separating shape from illumination, we can recognize an invariance of shape regardless of changing illumination.

The advantage of decomposing what we see into its more-or-less independent parts extends beyond the image formation process to the shapes and patterns on which that process operates. For instance, decomposing a bent rod into a straight rod and a bending transformation reveals the similarity between a bent rod and one that has not been bent, or some other solid that has been bent the same way.

Just as we need to understand the image-forming process to decompose an image into intrinsic images, we need to understand the processes that generate patterns to decompose them into intrinsic parts. But, while there is only one image-forming process, a staggering variety of processes shape and color the world around us. Our only hope of dealing with such complexity is to begin

with some basic pattern classes that recur in nature and understand how to decompose and describe them.

One such class consists of oriented patterns: patterns which everywhere have a dominant local orientation. Orientation selective mechanisms have been widely studied by physicists since Marr and Hildreth's discovery of orientation within the visual patterns [1997]. There has also been considerable interest among mathematicians in the perception of oriented patterns, particularly dot patterns. Only recently have the computational aspects received attention. Barr [1994] examined the grouping of tokens in satellite images based upon orientation and Horn [1997] investigated the estimation of orientation, by combining the outputs of linear operators in what he believes to be a biologically realistic manner.

Rather that modeling the visual system directly, we have focused on creating models of the types of physical processes, which give rise to commonly occuring visual patterns. Guided by these models, we have attempted to derive perceptual computations which measure the important physically significant parameters of the models.

In this project research, we consider oriented patterns such as those produced by propagation, accretion, or deformation. In order to recover the significant patterns, we must determine from the image: (1) what is propagating, accreting or deforming, and (2) which way and how much. More precisely, we must estimate everywhere the direction and magnitude of anisotropy (which we will call the flow field) and describe the residual pattern, independent of the field. Why this decomposition leads to simpler, more regular descriptions is best illustrated by example:

- A typical oriented pattern created by propagation is the streaked trail left by a paint brush dipped in variegated paint. The flow field describes the trajectory of the brush, while the residual pattern describes the distribution of paint on the brush.

- Accretion typically results in laminar structures, such as wood grain. Here, the flow field gives isochrones (the moving accretion boundary), and the residual pattern describes the change in color or brightness of the accreting material over time.

• If an isotropic body is deformed, the flow field principally describes the bending and stretching it has undergone, while the residual pattern describes the non-deformed body.

In all these cases, separate descriptions of the flow field and the residual pattern are appropriate because they describe different processes. The path of propagation for many physical processes is controlled by very different mechanisms that control the coloration of the trail left behind. Similarly, the mechanisms which control the shape of an accretion boundary are frequently unrelated to the processes controlling the color of the accreted material. Finally, the forces which deform a piece of material are often completely unrelated to the process which created the piece of material in the first place. By separately describing these processes, we can create descriptions of the whole which are often simpler than is possible without the separation, because each of the pieces may have different regularities.

In order to perform this separation, we propose an optimal algorithm for pattern thresholding. The first step is to examine local image neighborhoods and compute their dominant local orientation. These orientations can then be linked together into integral curves which everywhere are aligned with the computed orientation. For the types of physical processes just described, these curves correspond to propagation paths, accretion boundaries, or formerly straight lines which have been deformed.

For a number of image interpretation tasks, having the raw orientations or the integral curves may be sufficient, but for others it is important to know how neighboring integral lines are related. A systematic way to do this is to combine integral curves into a coordinate system. One coordinate tells you which integral curve you are on, while the other tells you where you are along that curve.

The local orientations, integral curves, and the coordinate system together completely specify the orientation structure or flow field of the image. If the original image intensities are viewed in the flow-field coordinate system, all the propagation paths, accretion boundaries, and deformed lines in the original are mapped to straight parallel lines. This simplified picture in which the changing orientation structure of the original has been removed, is what we mean by the residual pattern.

In the following section, a simple computation is developed for determining local orientations. Based on a nonlinear combination of image gradients, the computation is fast and reliable, provided the local pattern does not have more than one dominant orientation. In addition to an estimate of the dominant local

orientation, the computation provides a coherence measure which indicates the degree to which the local pattern is oriented. Several direct applications of these local measurements are provided including edge and anomaly detection.

## 4. Orientation Analysis

For intensity patterns created by anisotropic processes such as propagation, accretion, or deformation, variation in the flow direction is much slower than variation in the perpendicular direction (Leonard [1994]). Anisotropy in such patterns will be evident in the local power spectrum. The high frequency energy will tend to cluster along the line in the Fourier domain perpendicular to the flow orientation.

A simple way to detect this clustering is to sum the energy in an appropriate region of the power spectrum and examine how the sum is affected by rotations. This can be done by examining the energy in the output of an appropriate orientation-selective linear filter. The orientation at which the energy is maximum can be expected to be perpendicular to the flow orientation.

Selection of the filter involves a number of tradeoffs. Very low spatial frequencies are affected more strongly by illumination effects than surface coloration, so they are inappropriate for measuring textural anisotropy. Very high spatial frequencies are sensitive to noise and aliasing effects so they too are inappropriate. Hence some type of bandpass filtering is required.

The orientation specificity of the filter is also important. If the filter is too orientation-specific then a large neighborhood will be required in order to make reliable measurements of energy. Conversely, if the filter responds over a wide range of orientations then it will be difficult to localize the orientation very accurately. Thus there is a tradeoff between angular and spatial resolution. One reasonable choice for the frequency response is:

$$F(r, \theta) = [e^{-r^2 \sigma_1^2} - e^{-r^2 \sigma_2^2}]\, 2\pi i\, r\cos(\theta) \qquad (1)$$

The filter is bandpass with a band-width determined by $\sigma_1$ and $\sigma_2$. In our experience, ratios of sigmas in the range of 2.0 to 10 work well for vector

patterns. The orientation specificity (or tuning curve) is provided by the cosine dependence of the filter on $\theta$. This appears to strike a reasonable balance between angular and spatial resolutions for the range of patterns we have examined. The cosine orientation tuning curve of the filter has some unusually good properties for computing the filter output at different orientations. The impulse response $S\,(x,y)$ of the filter is

$$S\,(x,y) = \partial/\partial x\, H\,(x,y) \tag{2}$$

where

$$H\,(x,y) = [\sigma_1^{-2}\, e^{-r^2/\sigma_1^2} - \sigma_2^{-2}\, e^{-r^2/\sigma_2^2}] \tag{3}$$

is an isotropic filter. Let $C=H*I$ and let $R_\theta\,[S]$ denote a counterclockwise rotation of $S$ by an angle $\theta$ for the raster image $I$. Then the convolution $R_\theta\,[S] * I$ is just the directional derivative of $H*I$ in the $\theta$ direction. The directional derivative can be written in terms of the gradient $\wp$ so we have

$$R_\theta\,[S] * I = (\cos\theta, \sin\theta) \otimes \wp\,(H*I) \tag{4}$$

Thus a single convolution suffices for all orientations.

Since the filter $S$ severely attenuates very low frequencies, $R_\theta\,[S] * I$ can be safely regarded as zero mean. Thus the variance in its output can be estimated by the expression:

$$V(\theta) = W * (R_\theta\,[S] * I)^2 \tag{5}$$

Where $W\,(x,y)$ is a local weighting function with unit integral. We use a Gaussian weighting function $W\,(x,y)$ because approximate Gaussian convolutions can be computed without loss of generality.

Using the gradient formulation of the filter output shown in Equation (4), we can write the variance $V(\theta)$ as:

$$V(\theta) = W * (\cos\theta\, C_x + \sin\theta\, C_y)^2 \tag{6}$$

There remains the issue of interpreting $V(\theta)$. Assume that there is only one axis of anisotropy. Then $V(\theta)$ will have two extrema $\pi$ distance apart – corresponding to the major and minor axis. Let $V_2(\theta) = V(\theta/2)$. Then $V_2(\theta)$ will have a single extremum in the interval $0<\theta<2\pi$. A computationally inexpensive way of estimating the position of this extremum is to consider $V_2$ as a distribution and compute the mean. Since $V_2(\theta)$ is periodic, it should be considered as a distribution on the unit circle. Hence its mean is the vector integral $(\alpha,\beta) = \int V_2(\theta)(\cos\theta, \sin\theta)\, d\theta$ evaluated between zero and $2\pi$. The angle $\tan^{-1}(\beta/\alpha)$ is an estimate of the angle of the peak in $V_2$ and hence twice the angle of the peak in V. Thus the angle $\phi$ of greatest variance can be written

$$\phi = \tan^{-1}(\beta/\alpha) \tag{7a}$$

$$= (1/2)\tan^{-1}\left(\int V_2(\theta)\sin\theta\, d\theta \,/ \int V_2(\theta)\cos\theta\, d\theta\right) \tag{7b}$$

evaluated between zero and $2\pi$.

$$= (1/2)\tan^{-1}\left(\int V_2(\theta)\sin 2\theta\, d\theta \,/ \int V_2(2\theta)\cos\theta\, d\theta\right) \tag{7c}$$

evaluated between zero and $\pi$.

These integrals suggest that the angle of anisotropy $\phi$ can be written as:

$$\phi = (1/2)\tan^{-1}\left(W * 2\, C_x C_y \,/\, W * (C_x^2 - C_y^2)\right) \tag{8}$$

which directly yields a simple algorithm for computing $\phi$. Note that this computation is based on the assumption that there is a single axis of anisotropy. If more than one dominant orientation is present, the result will not be meaningful.

In addition to finding the direction of anisotropy, it is important to determine how strong an anisotropy there is. If the orientation of the local vector J is nearly uniformly distributed between zero and $2\pi$, then the orientation $\phi$ of slight anisotropy is not meaningful. Conversely, if all the J vectors are pointing in the same direction, then the indication of anisotropy is quite strong and $\phi$ is very meaningful. A simple way of measuring the strength of the peak in the distribution of J vectors is to look at the ratio:

$$X(x,y) = |W * J| / (W * |J|) \tag{9}$$

which we will call the coherence of the flow pattern. If the J vectors are close to being uniformly distributed, then the ratio will be nearly equal to zero. If the J vectors all point the same way, the ratio will be equal to unity. In between, the ratio will increase as the peak gets narrower.

## 5. Conclusions

We have seen that the orientation field is an abstraction from the anisotropic pattern that defines it. A single spiral field, for example, could arise from a pattern composed of multispectral bands or irregular topologies. The orientation field is useful because it describes the changing direction of anisotropy independent of the underlying pattern elements (background data) that define it.

Conversely, a description of the image in flow coordinates reveals the underlying pattern independent of the changing direction of anisotropy. Such a description would make it possible to recognize, for example, that two very different orientation fields are defined by the same bands. Intuitively, the orientation field describes the way the pattern is bent, while viewing the image in flow

coordinates straightens the pattern out. For many purposes, it is unnecessary to compute the deformed image explicitly, but doing so vividly illustrates how flow coordinates can simplify the pattern extraction (Williams [1998]).

Within this project research, we have addressed the problem of analyzing oriented patterns by decomposing them into a flow field, describing the direction of anisotropy, and describing the pattern independent of changing flow direction. A specific computation for estimating the flow direction was proposed. The computation can be viewed as: (a) finding the direction of maximal variance in the output of a linear filter, (b) combining gradient directions locally, and (c) finding the direction of maximum edge density. The flow field was then used to form a coordinate system to view the pattern. Two orthogonal families of curves – along and across the direction of flow – form the coordinate system's parameter lines. Viewing the pattern in these flow coordinates amounts to deforming the pattern so that the flow lines become parallel straight lines. The deformation produces a pattern that is simpler, more regular, and therefore more amenable to analysis. Within the next project report, specific case examples will be presented that describe this deformation process using multispectral and panchromatic data.

## 6. References

Barr, A., Global and Local deformations of Solid Primatives, J. Comput. Graphics, 18, 1994.

Horn, B. K. P., Understanding Image Intensities, J. Artif. Intell., 8, 1997.

Leonard, G., Database for Speaker-Independent Digit Recognition, Proceedings IEEE Int. Conf. Acoustics, Speech Signal Process, San Diego, Mar. 1994.

Marr, D. and E. Hildreth, Theory of Edge Detection, Proc. R. Soc. London, 207, 1997.

Williams, L., Pyramidal Parametrics, , J. Comput. Graphics, 22, 1998.

## 7. Appendix

The mathematical results shown in Equation (1-9) have been tested using C-language models for pattern recognition. A spiral variance field is defined using expression (6): $V(\theta) = W * (\cos\theta\, C_x + \sin\theta\, C_y)^2$. Gaussian vectors are then tested using: (a) purely orthogonal patterns, (b) ortho-normal patterns, and (c) discrete propagations. As in earlier project reports, the classes are selected, reduced, and combined using the geometric "spinning" of vectors in n-space. The complete simulation engine is programmed for Windows 95 and UNIX applications.

```
/*
 *   chars.VectorField.c
 *
 */

extern long fileSize;

char *dp;
char *countPtr;

AppendString(s)
register char *s;
{
     register char i, len;

     *countPtr += (len = *s++);
     for (i=0; i<len; i++) *dp++ = *s++;
}

#define CharOf(c)  ((((c)>=0x20)&&((c)<=0xCF))?(c):'.')

char hexChar[16] = {'0','1','2','3','4','5','6','7',
                             '8','9','A','B','C','D','E','F'};

DrawHex6(i)
register unsigned long i;
{
     register int mod = 20;
     register int j;
     register unsigned long mask = 0x000FFFFF;

     for (j=6;j>0; --j) {
          (*countPtr)++;
          *dp++ = (hexChar[i>>mod]);
          mod -= 4;
          i &= mask;
          mask >>= 4;
     }
}
```

```
DrawHex4(i)
register unsigned int i;
{
     register int mod = 12;
     register int j;
     register unsigned int mask = 0x0FFF;

     for (j=4;j>0; --j) {
            (*countPtr)++;
            *dp++ = hexChar[i>>mod];
            mod -= 4;
            i &= mask;
            mask >>= 4;
     }
}

Four_Blanks()
{
     (*countPtr) += 4;
     *dp++ = ' ';
     *dp++ = ' ';
     *dp++ = ' ';
     *dp++ = ' ';
}

Fill_Line(i, cptr, display)
long i;
register char    *cptr;
char *display;
{
     register int      k;
     register int      *ptr;
     register char     c;
     register long     n;

     ptr = (int *)cptr;

     dp = countPtr = display;
     *dp++=0;

     n = i;
     if (n >= fileSize) return;

     DrawHex6(i);
```

```
    AppendString("\p:  ");
    for (k=0;k<4;k++) {
          if (n>=fileSize)
                Four_Blanks();
          else
                DrawHex4(*ptr++);
          n += 2;
          (*countPtr)++;
          *dp++ = ' ';
    }

    (*countPtr)++;
    *dp++ = ' ';
    for (k=0;k<4;k++) {
          if (n>=fileSize)
                Four_Blanks();
          else
                DrawHex4(*ptr++);
          n += 2;
          (*countPtr)++;
          *dp++ = ' ';
    }
    AppendString("\p  ");

    n = i;
    for (k=0; k<16; k++) {
          c = *cptr++;
          if (++n>fileSize) return;
          (*countPtr)++;
          *dp++ = CharOf(c);
    }
}



/* buffer.Vector FieldDumpDA.c
 *
 *   Routines for managing a variance 'field' within the contents of a
 *   raster image.
 *
 *   SetBuffer(char *) set the pointer "buffer" to point to a given space
 *   FillBuffer( start, finish ) assure that the data from offset "start"
 *         to offset "finish" of the file is in the data buffer
 */

#include <FileMgr.h>
```

```
#define BUFSIZE  2048L
char             buffer[BUFSIZE];

extern    long       firstByte;
extern    int        nLines;
extern    int                  fileRef;

long lmax(a,b)
long a,b;
{
     return ((a>b) ? a : b);
}

#define PAD                    256L

ReadFrom(i)
long i;
{
     /* Maintains buffer and firstByte */
     /* Must assure that nLines of data are in buffer */
     /* If not:
     /*          firstByte = max(i-PAD, 0)
     /*          read BUFSIZE bytes at firstByte
     */

     long count;

     if ((i<firstByte) || (i + nLines*16>firstByte+BUFSIZE) || (firstByte<0)) {
           firstByte = lmax( i-PAD, 0L );
           SetFPos( fileRef, fsFromStart, firstByte );
           count = BUFSIZE;
           FSRead( fileRef, &count, &buffer );
     }
}



/* buffer.Creation of Variance Tools and Conditional Estimation Models.
 *
//
// Declared below are the module's 2 exported variables.
//
//   giNumVectorcubesThisProcess is an instance variable that contains
//   the number of (existing) Vectorcube controls created by the
//   current process.
//
//   giNumVectorcubesAllProcesses is a shared (between processes) variable
//   which contains the total number of (existing) Vectorcube controls
//   created by all processes in the system.
//
```

```
 *
 *  FUNCTION:    DllMain
 *
 *  INPUTS:      hDLL        - DLL module handle
 *               dwReason    - reason being called (e.g. process attaching)
 *               lpReserved - reserved
 *
 *  RETURNS:     TRUE if initialization passed, or
 *               FALSE if initialization failed.
 *
 *  COMMENTS:    On DLL_PROCESS_ATTACH registers the VECTORCUBECLASS
 *
 *               DLL initialization serialization is guaranteed within a
 *               process (if multiple threads then DLL entry points are
 *               serialized), but is not guaranteed across processes.
 *
 *               When synchronization objects are created, it is necesaary
 *               to check the return code of GetLastError even if the create
 *               call succeeded. If the object existed, ERROR_ALREADY_EXISTED
 *               will be returned.
 *
 *               If your DLL uses any C runtime functions then you should
 *               always call _CRT_INIT so that the C runtime can initialize
 *               itself appropriately. Failure to do this may result in
 *               indeterminate behavior. When the DLL entry point is called
 *               for DLL_PROCESS_ATTACH & DLL_THREAD_ATTACH circumstances,
 *               _CRT_INIT should be called before any other initilization
 *               is performed. When the DLL entry point is called for
 *               DLL_PROCESS_DETACH & DLL_THREAD_DETACH circumstances,
 *               _CRT_INIT should be called after all cleanup has been
 *               performed, i.e. right before the function returns.
 *
```

```
BOOL WINAPI DllMain (HANDLE hDLL, DWORD dwReason, LPVOID lpReserved)
{
  ghMod = hDLL;
  switch (dwReason)
  {
    case DLL_PROCESS_ATTACH:
    {
      WNDCLASS wc;

      if (!_CRT_INIT (hDLL, dwReason, lpReserved))

        return FALSE;

      wc.style          = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS | CS_OWNDC |
                          CS_GLOBALCLASS ;
      wc.lpfnWndProc    = (WNDPROC) VectorcubeWndProc;
      wc.cbClsExtra     = 0;
      wc.cbWndExtra     = VECTORCUBE_EXTRA;
      wc.hInstance      = hDLL;
      wc.hIcon          = NULL;
      wc.hCursor        = LoadCursor (NULL, IDC_ARROW);
      wc.hbrBackground = NULL;
      wc.lpszMenuName  = (LPSTR) NULL;
      wc.lpszClassName = (LPSTR) VECTORCUBECLASS;

      if (!RegisterClass (&wc))
      {
        MessageBox (NULL,
                    (LPCTSTR) "DllMain(): RegisterClass() failed",
                    (LPCTSTR) "Err! - VECTORCUBE.DLL",
                    MB_OK | MB_ICONEXCLAMATION);

        return FALSE;
      }

      break;
    }
```

```
    case DLL_PROCESS_DETACH:
    {

      if (!_CRT_INIT (hDLL, dwReason, lpReserved))

        return FALSE;

      if (!UnregisterClass ((LPSTR) VECTORCUBECLASS, hDLL ))
      {
        MessageBox (NULL,
                    (LPCTSTR) "DllMain(): UnregisterClass() failed",
                    (LPCTSTR) "Err! - VECTORCUBE.DLL",
                    MB_OK | MB_ICONEXCLAMATION);

        return FALSE;
      }

      break;
    }


    default:

      if (!_CRT_INIT (hDLL, dwReason, lpReserved))

        return FALSE;

      break;
  }
  return TRUE;
}


/************************************************************************\
*
*  FUNCTION:    CustomControlInfoA
*
*  INPUTS:      acci - pointer to an array od CCINFOA structures
*
*  RETURNS:     Number of controls supported by this DLL
*
*  COMMENTS:    See CUSTCNTL.H for more info
*
\************************************************************************/
```

```
UINT CALLBACK CustomControlInfoA (LPCCINFOA acci)
{
  //
  // Dlgedit is querying the number of controls this DLL supports, so return 1.
  //   Then we'll get called again with a valid "acci"
  //

  if (!acci)

    return 1;


  //
  // Fill in the constant calues.
  //

  acci[0].flOptions           = 0;
  acci[0].cxDefault           = 40;      // default width  (vector units)
  acci[0].cyDefault           = 40;      // default height (vector units)
  acci[0].flStyleDefault      = WS_CHILD |
                                WS_VISIBLE |
                                SS_INMOTION;
  acci[0].flExtStyleDefault = 0;
  acci[0].flCtrlTypeMask      = 0;
  acci[0].cStyleFlags         = NUM_VECTORCUBE_STYLES;
  acci[0].aStyleFlags         = aVectorcubeStyleFlags;
  acci[0].lpfnStyle           = VectorcubeStyle;
  acci[0].lpfnSizeToText      = VectorcubeSizeToText;
  acci[0].dwReserved1         = 0;
  acci[0].dwReserved2         = 0;


  //
  // Copy the strings (Segmented Vector Measurements within a "string" data
structure - Very
  //   Compressed!
  //
  // NOTE: MAKE SURE THE STRINGS COPIED DO NOT EXCEED THE LENGTH OF
  //       THE BUFFERS IN THE CCINFO STRUCTURE!
  //

  lstrcpy (acci[0].szClass, VECTORCUBECLASS);
  lstrcpy (acci[0].szDesc,  VECTORCUBEDESCRIPTION);
  lstrcpy (acci[0].szTextDefault, VECTORCUBEDEFAULTTEXT);


  //
  // Return the number of controls that the DLL supports
  //

  return 1;
}
```

```
 *
 *  FUNCTION:     VectorcubeStyle
 *
 *  INPUTS:       hWndParent - handle of parent window (dialog editor)
 *                pccs       - pointer to a CCSTYLE structure
 *
 *  RETURNS:      TRUE  if success,
 *                FALSE if error occured
 *
 *  LOCAL VARS:   rc - return code from DialogBox
 *
\*****************************************************************************/

BOOL CALLBACK VectorcubeStyle (HWND hWndParent, LPCCSTYLE pccs)
{
  int rc;

  gpccs = pccs;

  if ((rc = DialogBox (ghMod, "VectorcubeStyle", hWndParent,
                       (DLGPROC)VectorcubeDlgProc)) == -1)
  {
    MessageBox (hWndParent, (LPCTSTR) "VectorcubeStyle(): DialogBox failed",
                (LPCTSTR) "Err!- Vectorcube.dll",
                MB_OK | MB_ICONEXCLAMATION | MB_APPLMODAL);
    rc = 0;
  }

  return (BOOL) rc;
}


/*****************************************************************************\
 *
 *  FUNCTION:     VectorcubeSizeToText
 *
 *  INPUTS:       flStyle     - control style
 *                flExtStyle - control extended style
 *                hFont       - handle of font used to draw text
 *                pszText     - control text
 *
 *  RETURNS:      Width (in pixels) control must be to accomodate text, or
 *                -1 if an error occurs.
 *
 *  COMMENTS:     Just no-op here (since we never actually display text in
 *                the control it doesn't need to be resized).
 *
\*****************************************************************************/
```

```
INT CALLBACK VectorcubeSizeToText (DWORD flStyle, DWORD flExtStyle,
                                  HFONT hFont,    LPSTR pszText)
{
  return -1;
}



/*****************************************************************************\
*
*  FUNCTION:     VectorcubeWndProc (standard window procedure INPUTS/RETURNS)
*
*  COMMENTS:     This is the window procedure for our custom control. At
*                creation we alloc a VECTORCUBEINFO struct, initialize it,
*                and associate it with this particular control. We also
*                start a timer which will invalidate.).
*
\*****************************************************************************/

LRESULT CALLBACK VectorcubeWndProc (HWND hwnd, UINT msg, WPARAM wParam,
                                   LPARAM lParam)
{
  switch (msg)
  {
    case WM_CREATE:
    {
      //
      // Alloc & init a VECTORCUBEINFO struct for this particular control
      //

      HDC             hdc;
      LPCREATESTRUCT lpcs = (LPCREATESTRUCT) lParam;
      PVECTORCUBEINFO  pSCI = (PVECTORCUBEINFO) LocalAlloc (LPTR,
                                                sizeof(VECTORCUBEINFO));
      if (!pSCI)
      {
        MessageBox (NULL,
                    (LPCTSTR) "VectorcubeWndProc(): LocalAlloc() failed",
                    (LPCTSTR) "Err! - VECTORCUBE.DLL",
                    MB_OK | MB_ICONEXCLAMATION);
        return -1;
      }


      //
      // Alloc the compatible DC for this control.
      //

      hdc = GetDC (hwnd);

      if ((pSCI->hdcCompat = CreateCompatibleDC (hdc)) == NULL)
      {
        MessageBox (NULL,
                    (LPCTSTR) "VectorcubeWndProc(): CreateCompatibleDC() failed",
```

```
              (LPCTSTR) "Err! - VECTORCUBE.DLL",
              MB_OK | MB_ICONEXCLAMATION);
  return -1;
}

ReleaseDC (hwnd, hdc);


//
// Initialize this instance structure
//

pSCI->fCurrentXRotation =
pSCI->fCurrentYRotation =
pSCI->fCurrentZRotation = (float) 0.0;

pSCI->fCurrentXRotationInc =
pSCI->fCurrentYRotationInc =
pSCI->fCurrentZRotationInc = (float) 0.2617; // a random # (15 degrees)

pSCI->iCurrentXTranslation =
pSCI->iCurrentYTranslation =
pSCI->iCurrentZTranslation = 0;

//
// All these calculations so the cubes start out with random movements.
//

if ((pSCI->iCurrentXTranslationInc = (rand() % 10) + 2) > 7)

  pSCI->iCurrentXTranslationInc = -pSCI->iCurrentXTranslationInc;

if ((pSCI->iCurrentYTranslationInc = (rand() % 10) + 2) <= 7)

  pSCI->iCurrentYTranslationInc = -pSCI->iCurrentYTranslationInc;

if ((pSCI->iCurrentZTranslationInc = (rand() % 10) + 2) > 7)

  pSCI->iCurrentZTranslationInc = -pSCI->iCurrentZTranslationInc;

pSCI->rcCubeBoundary.left   =
pSCI->rcCubeBoundary.top    = 0;
pSCI->rcCubeBoundary.right  = lpcs->cx;
pSCI->rcCubeBoundary.bottom = lpcs->cy;
pSCI->iOptions  = VECTORCUBE_REPAINT_BKGND;
pSCI->hbmCompat = NULL;

SetWindowLong (hwnd, GWL_VECTORCUBEDATA, (LONG) pSCI);

SetTimer (hwnd, VECTOR_EVENT, VECTOR_INTERVAL, NULL);

//
// Increment the count vars
//
```

```
    giNumVectorcubesThisProcess++;
    giNumVectorcubesAllProcesses++;

    break;
  }

  case WM_PAINT:

    Paint (hwnd);
    break;

  case WM_TIMER:

    switch (wParam)
    {
      case VECTOR_EVENT:
      {
        PVECTORCUBEINFO pSCI = (PVECTORCUBEINFO) GetWindowLong (hwnd,
                                                     GWL_VECTORCUBEDATA);

        InvalidateRect (hwnd, &pSCI->rcCubeBoundary, FALSE);

        break;
      }
    }

    break;

  case WM_LBUTTONDBLCLK:
  {
    //
    // Toggle the erase state of the control
    //

    if (DO_ERASE(hwnd))

      SetWindowLong (hwnd, GWL_STYLE,
                     GetWindowLong (hwnd, GWL_STYLE) & ~SS_ERASE);


    else
    {
      //
      // Repaint the entire control to get rid of the (cube trails) mess
      //

      PVECTORCUBEINFO pSCI = (PVECTORCUBEINFO) GetWindowLong (hwnd,
                                                     GWL_VECTORCUBEDATA);

      SetWindowLong (hwnd, GWL_STYLE,
                     GetWindowLong (hwnd, GWL_STYLE) | SS_ERASE);
      pSCI->iOptions |= VECTORCUBE_REPAINT_BKGND;
      InvalidateRect (hwnd, NULL, FALSE);
      SendMessage (hwnd, WM_PAINT, 0, 0);
    }
```

```
      break;
    }

    case WM_RBUTTONDBLCLK:
    {
      //
      // Toggle the motion state of the control
      //

      if (IN_MOTION(hwnd))
      {
        KillTimer (hwnd, VECTOR_EVENT);
        SetWindowLong (hwnd, GWL_STYLE,
                       GetWindowLong (hwnd, GWL_STYLE) & ~SS_INMOTION);
      }
      else
      {
        SetTimer (hwnd, VECTOR_EVENT, VECTOR_INTERVAL, NULL);
        SetWindowLong (hwnd, GWL_STYLE,
                       GetWindowLong (hwnd, GWL_STYLE) | SS_INMOTION);
      }

      break;
    }

    case WM_SIZE:

      if (wParam == SIZE_MAXIMIZED || wParam == SIZE_RESTORED)
      {
        PVECTORCUBEINFO pSCI = (PVECTORCUBEINFO) GetWindowLong (hwnd,
                                                   GWL_VECTORCUBEDATA);
        //
        // Get a new bitmap which is the new size of our window
        //

        HDC hdc = GetDC (hwnd);
        HBITMAP hbmTemp = CreateCompatibleBitmap (hdc,
                                                  (int) LOWORD (lParam),
                                                  (int) HIWORD (lParam));
        if (!hbmTemp)
        {
          //
          // Scream, yell, & committ an untimely demise...
          //

          MessageBox (NULL,
                      (LPCTSTR)  "VectorcubeWndProc():  CreateCompatibleBitmap()
failed",
                      (LPCTSTR) "Err! - VECTORCUBE.DLL",
                      MB_OK | MB_ICONEXCLAMATION);
          DestroyWindow (hwnd);
        }

        pSCI->hbmSave = SelectObject (pSCI->hdcCompat, hbmTemp);
        if (pSCI->hbmCompat)
```

```
        DeleteObject (pSCI->hbmCompat);
      ReleaseDC    (hwnd, hdc);
      pSCI->hbmCompat = hbmTemp;


      //
      // Reset the translation so the cube doesn't go vectorning off into
      //    space somewhere- we'd never see it again!
      //

      pSCI->iCurrentXTranslation =
      pSCI->iCurrentYTranslation =
      pSCI->iCurrentZTranslation = 0;

      //
      // All these calculations so the cube starts out with random movements,
      //

      if ((pSCI->iCurrentXTranslationInc = (rand() % 10) + 2) > 7)

        pSCI->iCurrentXTranslationInc = -pSCI->iCurrentXTranslationInc;

      if ((pSCI->iCurrentYTranslationInc = (rand() % 10) + 2) <= 7)

        pSCI->iCurrentYTranslationInc = -pSCI->iCurrentYTranslationInc;

      if ((pSCI->iCurrentZTranslationInc = (rand() % 10) + 2) > 7)

        pSCI->iCurrentZTranslationInc = -pSCI->iCurrentZTranslationInc;

      pSCI->rcCubeBoundary.left   =
      pSCI->rcCubeBoundary.top    = 0;
      pSCI->rcCubeBoundary.right  = (int) LOWORD (lParam);
      pSCI->rcCubeBoundary.bottom = (int) HIWORD (lParam);

      pSCI->iOptions |= VECTORCUBE_REPAINT_BKGND;

      InvalidateRect (hwnd, NULL, FALSE);
    }

    break;

  case WM_DESTROY:
  {
    PVECTORCUBEINFO pSCI = (PVECTORCUBEINFO) GetWindowLong (hwnd,
                                                    GWL_VECTORCUBEDATA);
    //
    // Clean up all the resources used for this control
    //

    if (IN_MOTION(hwnd))

      KillTimer (hwnd, VECTOR_EVENT);

    SelectObject (pSCI->hdcCompat, pSCI->hbmSave);
```

```
            DeleteObject (pSCI->hbmCompat);
            DeleteDC     (pSCI->hdcCompat);

            LocalFree (LocalHandle ((LPVOID) pSCI));


            //
            // Decrement the global count vars
            //

            giNumVectorcubesThisProcess--;
            giNumVectorcubesAllProcesses--;

            break;
        }

        default:

            return (DefWindowProc(hwnd, msg, wParam, lParam));
    }

    return ((LONG) TRUE);
}
```

```
\****************************************************************************/
*
*  FUNCTION:     VectorcubeDlgProc (standard dialog procedure INPUTS/RETURNS)
*
*  COMMENTS:     This dialog comes up in response to a user requesting to
*                modify the control style. This sample allows for changing
*                the control's text, and this is done by modifying the
*                CCSTYLE structure pointed at by "gpccs" (a pointer
*                that was passed to us by dlgedit).
*
\****************************************************************************/

LRESULT CALLBACK VectorcubeDlgProc (HWND hDlg, UINT msg, WPARAM wParam,
                                  LPARAM lParam)
{
  switch (msg)
  {
    case WM_INITDIALOG :
    {
      if (gpccs->flStyle & SS_ERASE)

        CheckDlgButton (hDlg, DID_ERASE, 1);

      if (gpccs->flStyle & SS_INMOTION)

        CheckDlgButton (hDlg, DID_INMOTION, 1);

      break;
    }

    case WM_COMMAND:

      switch (LOWORD(wParam))
      {
        case DID_ERASE:

          if (IsDlgButtonChecked (hDlg, DID_ERASE))

            gpccs->flStyle |= SS_ERASE;

          else

            gpccs->flStyle &= ~SS_ERASE;

          break;

        case DID_INMOTION:

          if (IsDlgButtonChecked (hDlg, DID_INMOTION))

            gpccs->flStyle |= SS_INMOTION;

          else

            gpccs->flStyle &= ~SS_INMOTION;
```

```
                break;

            case DID_OK:

                EndDialog  (hDlg, 1);
                break;
        }
        break;
    }
    return FALSE;
}
```

```
/*****
 * vectMenus.c
 *
 *   Routines for Vectfield menus.
 *
 *****/

extern      WindowPtr vectfieldWindow;
extern      int    width;


MenuHandle unixMenu, fileMenu, editMenu, widthMenu;

enum {
     unixID = 1,
     fileID,
     editID,
     widthID
     };

enum {
     openItem = 1,
     closeItem,
     quitItem = 4
     };

/****
 * SetUpMenus()
 *
 *   Set up the menus. Normally, we'd use a resource file, but
 *   for this example we'll supply "hardwired" strings.
 *
 ****/

SetUpMenus()

{
     InsertMenu(unixMenu = NewMenu(unixID, "\p\024"), 0);
```

```
     InsertMenu(fileMenu = NewMenu(fileID, "\pFile"), 0);
     InsertMenu(editMenu = NewMenu(editID, "\pEdit"), 0);
     InsertMenu(widthMenu = NewMenu(widthID, "\pWidth"), 0);
     DrawMenuBar();
     AddResMenu(unixMenu, 'DRVR');
     AppendMenu(fileMenu, "\pOpen/O;Close/W;(-;Quit/Q");
     AppendMenu(editMenu, "\pUndo/Z;(-;Cut/X;Copy/C;Paste/V;Clear");
     AppendMenu(widthMenu, "\p1/1;2/2;3/3;4/4;5/5;6/6;7/7;8/8;9/9;10/0;11;12");
}
/* end SetUpMenus */


/****
 *  AdjustMenus()
 *
 *   Enable or disable the items in the Edit menu if a DA window
 *   comes up or goes away. Our application doesn't do anything with
 *   the Edit menu.
 *
 ****/

AdjustMenus()
{
     register WindowPeek wp = (WindowPeek) FrontWindow();
     short kind = wp ? wp->windowKind : 0;
     Boolean DA = kind < 0;

     enable(editMenu, 1, DA);
     enable(editMenu, 3, DA);
     enable(editMenu, 4, DA);
     enable(editMenu, 5, DA);
     enable(editMenu, 6, DA);

     enable(fileMenu, openItem, !((WindowPeek) vectfieldWindow)->visible);
     enable(fileMenu, closeItem, DA || ((WindowPeek) vectfieldWindow)->visible);

     CheckItem(widthMenu, width, true);
}


static
enable(menu, item, ok)
Handle menu;
{
     if (ok)
           EnableItem(menu, item);
     else
           DisableItem(menu, item);
}


/*****
 * HandleMenu(mSelect)
 *
 *   Handle the menu selection. mSelect is what MenuSelect() and
```

```
 *   MenuKey() return: the high word is the menu ID, the low word
 *   is the menu item
 *
 *****/

HandleMenu (mSelect)

     long  mSelect;

{
     int                  menuID = HiWord(mSelect);
     int                  menuItem = LoWord(mSelect);
     Str255               name;
     GrafPtr              savePort;
     WindowPeek  frontWindow;

     switch (menuID)
       {
       case      unixID:
            GetPort(&savePort);
            GetItem(unixMenu, menuItem, name);
            OpenDeskAcc(name);
            SetPort(savePort);
            break;

       case      fileID:
            switch (menuItem)
              {
              case      openItem:
                   ShowWindow(vectfieldWindow);
                   SelectWindow(vectfieldWindow);
                   break;

              case      closeItem:
                   if ((frontWindow = (WindowPeek) FrontWindow()) == 0L)
                     break;

                   if (frontWindow->windowKind < 0)
                     CloseDeskAcc(frontWindow->windowKind);
                   else if (frontWindow = (WindowPeek) vectfieldWindow)
                     HideWindow(vectfieldWindow);
                                break;

              case      quitItem:
                   ExitToShell();
                   break;
              }
            break;

       case      editID:
            if (!SystemEdit(menuItem-1))
              SysBeep(5);
            break;

       case      widthID:
```

```
            CheckItem(widthMenu, width, false);
            width = menuItem;
            InvalRect(&vectfieldWindow->portRect);
            break;
        }
}
/* end HandleMenu */
```

Annex to

**Third Interim Report (07 July 1998)**
**Conditional Estimation of Vector Patterns in Remote Sensing and GIS**

**contract no. N 68171 97 C 9027**
**contractor: UvA, Applied Logic Labratory/CCSOM**
**Principal Investigator: Dr. M. Masuch**

**1. Statement showing amount of unused funds at the end of the covered period**

| | | |
|---|---|---|
| 2nd Incrementally Funded Peric total<br>December 1998 –September 1999 | $ | 150,000.00 |
| 3rd Incrementally Funded Perio total<br>October 1999 - July 2000 | $ | 150,000.00 |
| **Total unused funds from June 1997 until Februari 2** | $ | 300,000 |

**2. List of impotant property acquired with contract funds during this period**

none